Aspekt- und subjektorientierte Programmierung

Michael Stürmer
Hauptseminar Objektorientierter Entwurf
E-mail: ms@mallorn.de

Zusammenfassung

Die heute vorhandenen komplexen und langlebigen Software-System sind immer schwieriger zu entwickeln und zu warten. Der Code ist verwickelt, die verschiedenen Anwendungen durch das gesamte System verstreut. Im folgenden werden verschiedene Lösungsansätze vorgestellt, die diese Probleme durch die Kapselung und Modularisierung der Anforderungen beheben wollen.

1 Einführung

In der gesamten Geschichte der Softwareentwicklung gab es immer wieder die Schwierigkeit, Software so zu schreiben, daß sie erweiterbar und wartbar bleibt. Eine erste Technik war die prozedurorientierte Programmierung, bei der verschiedene Funktionen als modulare, teilweise wiederverwendbare Prozeduren gekapselt werden. Auch bei der objektorientierten Programmierung wird versucht, Objekte und deren Verhalten zu kapseln. Diese Techniken sind teilweise erfolgreich.

Da Software heutzutage immer langlebiger und weiter verbreitet wird, und immer öfter verschiedene Softwaresysteme in ein Produkt integriert werden sollen, reichen die bisherigen Ausdrucksmittel nicht aus. Der Code wird verwickelt (*code tangling*).

1.1 konventionelle Lösungen

Eine Möglichkeit, diesem Problem zu begegnen, ist das Refactoring: Der Code muß für die neuen Anforderungen umstrukturiert werden. In der OOP können neue Anforderungen an eine Klasse z.B. in Unterklassen implementiert werden. Damit sind sie von der ursprünglichen Implementierung gekapselt. Eine weitere Möglichkeit ist der Einsatz von Design Patterns, die den Code flexibler machen und somit das Einführen neuer Funktionalität vereinfacht. Allerdings löst das nicht alle Probleme.

Obwohl der Code gekapselt ist, ist er weiterhin verstreut. Es ist eher schwieriger, den Code zu verstehen,

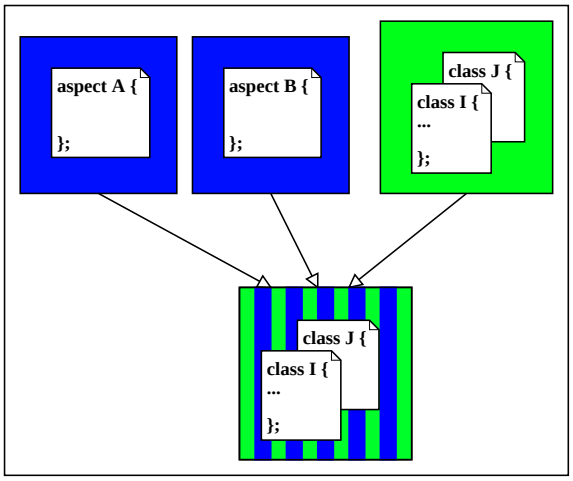

Abbildung 1: aspect weaver

z.B. wenn das Design Pattern nicht bekannt ist. Außerdem sind nicht vorausgeplante Änderungen weiterhin schwierig und erfordern jedes Mal neues Refactoring.

2 Aspektorientierte Programmierung (AOP)

Die AOP[5] ist ein anderer Ansatz: die verschiedenen überkreuzenden Anliegen (*crosscutting concerns*) sollen modularisiert und an einer Stelle gekapselt werden. Diese Einheit wird Aspekt genannt. Sie sollen beliebig, je nach Anforderung kombinierbar sein. Der Vorgang der Kombination wird weben genannt und mit einem *aspect weaver* durchgeführt (siehe Abb. 1).

Durch diese Aufteilung wird der Code einfacher lesbar und wartbar, er hat eine höhere Lokalität. Außerdem ist die Wiederverwendung einfacher. Ein ganzer Aspekt kann einfach wiederverwendet werden.

2.1 Weaving

Nach [2] gibt es mehrere Möglichkeiten, die verschiedenen Aspekte zu verweben. Eine Technik ist das statische *Weaving*. Dabei wird zur Kompilierzeit an be-

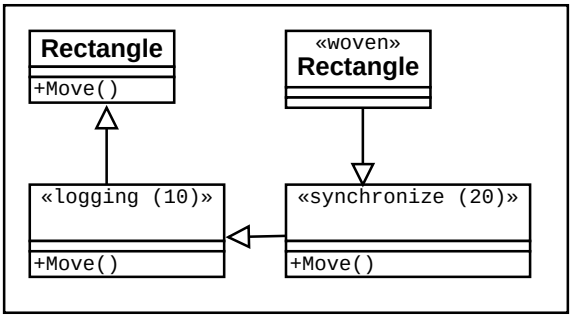

Abbildung 2: dynamisches Weaving

stimmten Stellen im Code, den sog. *Join Points*, Code eingefügt, so wie es im Aspekt definiert wurde. Dies hat keine Verlangsamung der Ausführung zur Laufzeit zur Folge, jedoch ist es zur Laufzeit auch nicht möglich, das Verhalten zu ändern.

Beim dynamischen *Weaving* ist das möglich. Dabei werden für jeden Aspekt zusätzliche Klassen angelegt, in denen das Verhalten gekapselt ist. Zusätzlich ist eine Dispatcher-Klasse notwendig, die je nach Konfiguration bei einem Methodenaufruf zusätzlich die entsprechenden Aspektklassen aufruft. In Abb. 2 ist das am Beispiel einer Rectangle-Klasse skizziert.

Dieses dynamische Weaving kann z.B. dazu benutzt werden, um Tracing bei Problemen zur Laufzeit einschalten. Ein weiteres Beispiel wäre ein Load-Balancer Aspekt, der durch einen anderen mit einer neuen Strategie ersetzt werden soll.

3 Realisierungen

Es gibt verschiedene Möglichkeiten, Aspektorientierung umzusetzen. Einmal auf Implementierungsebene, mit Sprachen, die allgemeine Sprachkonstrukte besitzen, um Aspekte auszudrücken. Dazu gehören AspectJ und HyperJ, die nachher noch genauer dargestellt werden. Eine weitere Möglichkeit ist die Erweiterung einer konventionellen Sprache um eigene Konstrukte. Dies bringt bei speziellen Anforderungen wie z.B. Synchronisation Vorteile, da sie sehr speziell angepasst werden können.

Des weiteren gibt es auch Frameworks, die es ermöglichen, Aspekte auszudrücken. Diese sorgen dann, z.B. mit Hilfe einer Funktionalität wie Reflektion, dafür, daß Aspekt-Code zum richtigen Zeitpunkt aufgerufen wird.

Unter Performance–Gesichtspunkten ist das Benutzen einer Aspektsprache vorzuziehen, da das Verweben zur Kompilierzeit stattfindet. Bei einem Framework ist zur Laufzeit zusätzlicher Aufwand notwendig.

Allerdings ist schwierig, für eine um Aspekte erweiterte Sprache Werkzeuge zu finden. Die bisher benutzen Werkzeuge und Entwicklungsumgebungen können teilweise nicht weiterverwendet werden.

Außerdem ist es bei einigen Aspektsprachen nur eingeschränkt möglich, die Aspekte zu erweitern, oder neue Aspekte einzuführen, so daß dann evtl. auch ein Refactoring notwendig wird. Demgegenüber sind Frameworks flexibler, allerdings auch nur in dem Ausmaß, das die Frameworkentwickler vorausgeplant haben.

4 AspectJ

AspectJ[6][1] ist ein um aspektorientierte Sprachkonstrukte erweiterter Java-Dialekt.

Damit AspectJ einfach angenommen wird, wurde beim Design sehr auf Kompatibilität geachtet:

- Aufwärts-Kompatibilität

  alle zulässigen Java-Programme sind auch zulässiges AspectJ

- Plattform-Kompatibilität

  zulässiges AspectJ läuft in der Standard-JVM

- Werkzeug-Kompatibilität

  Werkzeuge sollen AspectJ „natürlich“ unterstützen

- Programmierer-Kompatibilität

  AspectJ soll sich wie eine natürliche Java-Erweiterung anfühlen

AspectJ ist ein praktischer Ansatz, der in der Industrie eingesetzt werden kann, es ist nicht nur als Spielzeug zum Ausprobieren neuer Ideen entstanden.

4.1 *Join Points*

Das einzige neu eingeführte Konzept in AspectJ sind die Verbindungspunkte *Join Points* [6, 3.1]. Ein Join Point ist ein eindeutig definierter Punkt im Programmablauf. Allerdings nicht *überall* im Programmcode, so ist ein Join Point wie „in Zeile 17 in Test.java“ nicht möglich. Dies wurde eingeschränkt, um die Lesbarkeit und das Verständnis des Codes nicht einzuschränken.

Folgende Stellen im Programmablauf sind Join Points:

Methoden-Aufruf, Konstruktor-Aufruf Empfang des Methoden-Aufrufs, des Konstruktor-Aufrufs Methoden-Ausführung, Konstruktor-Ausführung
Lesen eines Felds Setzen eines Felds
Exception-Handler-Ausführung Klassen-Initialisierung Objekt-Initialisierung

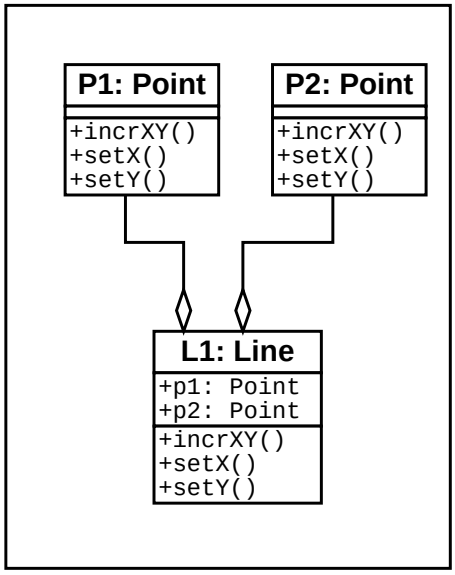

Abbildung 3: Join Points

Als Beispiel soll folgender Quelltext dienen, die entstandenen Objekte werden in Abb. 3 verdeutlicht:

```
Point P1 = new Point(0, 0);
Point P2 = new Point(1, 1);
Line  L1 = new Line(P1, P2);

L1.incrXY(1, 1);
```

Folgende Join Points werden beim Methoden-Aufruf durchlaufen:

1. Ein Methoden-Aufruf beim Aufruf von incrXY im Objekt L1
2. Empfang des Methoden-Aufrufs: L1 empfängt den Aufruf von incrXY
3. Methoden-Ausführung: incrXY wird ausgeführt
4. Lesen des Felds p1 in L1
5. Methoden-Aufruf beim Aufruf von incrXY im Objekt P1, ...

Die ganze Kette wird dann beim Rücksprung der Methode rückwärts durchlaufen.

4.2 *Pointcut Designators*

Pointcut Designators werden kurz auch „Pointcuts“ genannt. Ein Pointcut Designator ist eine Menge von *Join Point*s und evtl. Werte aus dem Kontext des Join Points.

Ein Join Point wird auf diese Weise definiert:

```
call (void Point.setXY (int, int))
```

Dies ist lediglich eine Definition für einen Zeitpunkt im Programmablauf: der Zeitpunkt, an dem die Methode setXY in der Klasse Point aufgerufen wird. Genau wie bei einer Typdefinition *geschieht* dadurch nichts. Der Einsatz dieses Konstrukts wird später gezeigt.

Pointcuts können auch mit Operatoren wie &&, `||` und `!` verbunden werden:

```
call (void Point.setX (int)) ||
call (void Point.setY (int))
```

Desweiteren ist es möglich, Pointcuts zu benennen. Diese können dann später in Advices benutzt werden.

```
pointcut move():
  call (void Point.setX (int)) ||
  call (void Point.setY (int));
```

Bisher wurden nur namensbasierte Pointcuts vorgestellt. Es gibt aber auch die Möglichkeit, Pointcuts mit Hilfe von Eigenschaften zu definieren:

```
call (void Figure.make*(..))
  && cflow (move())
```

Dieser Pointcut enthält alle Join Points, die Methodenaufrufen an die Klasse `Figure` sind, die mit `make` anfangen. Außerdem müssen die Join Points während der Ausführung eines der Join Points im Pointcut `move()` liegen.

Schließlich ist es auch möglich, auf den Kontext des Pointcuts zuzugreifen:

```
pointcut setXY (FigureElement fe,
                int           x,
                int           y):

call (void fe.setXY(x, y))
```

4.3 *Advices*

Ein Advice ist ein Codefragment, das beim Erreichen eines bestimmten Join Points ausgeführt wird. Es gibt drei verschiedene Advices:

- before advice

  Der Code wird vor dem Join Point ausgeführt

- around advice

  Der Code wird vor und nach dem Join Point ausgeführt

- after advice

  Der Code wird nach dem Join Point ausgeführt

Ein einfaches Beispiel für einen after-Advice wäre:

```
after (): moves() {
  flag = true;
}
```

Durch diese Deklaration wird `flag = true` ausgeführt, nachdem die Programmausführung „auf dem Rückweg“ an einem Join Point im Pointcut `moves()` vorbeikommt.

Wie vorhin schon angedeutet, ist auch der Zugriff auf den Kontext eines Join Points möglich:

```
after (FigureElement fe,
       int           x,
       int           y):
  setXY (fe, x, y) {
    System.out.println
      (fe+" moved to "+x+", "+y);
  }
```

Es gibt eine festgelegte Reihenfolge, in der Advices am Join Point ausgeführt werden:

- Sind zwei Advices im gleichen Aspekt definiert, werden sie in der Reihenfolge ausgeführt, in der sie definiert wurden.
- Erweitert ein Aspekt einen anderen, übernimmt der speziellere Aspekt den Advice aus dem allgemeineren. Advices im spezielleren Aspekt haben Vorrang.
- Es gibt ein `dominates` Schlüsselwort, mit dem man angeben kann, welcher Aspekt Vorrang hat.
- Ansonsten ist die Reihenfolge undefiniert, z.B. wenn die Aspekte völlig unabhängig voneinander sind.

4.4 *Introduction*

Mittels einer Introduction kann die statische Typsignatur einer existierenden Klasse geändert werden. Dies wird benutzt, um eine Klasse um neue Methoden oder Felder zu erweitern, um Vererbungsbeziehungen zu ändern oder neue Interfaces implementieren.

4.5 *Aspects*

Die bisher vorgestellten Konstrukte machen für sich gesehen keinen Sinn. Sie müssen innerhalb eines Aspekts benutzt werden. Aspekte kapseln dann die Implementierungen der verschiedenen Anwendungen.

In AspectJ ist ein Aspekt ein neues Sprachelement, das viele Gemeinsamkeiten mit Klassen hat:

- typisiert
- erben von Klassen, Aspekten
- abstrakt oder konkret
- können instantiiert werden
- haben statisches und dynamisches Verhalten
- haben Felder, Methoden und Typen

Natürlich gibt es auch einige Unterschiede:

- Definition von Pointcuts, Advices und Introductions
- nicht als Klasse verwendbar
- keine Konstruktoren, Destruktoren

Im folgenden werden einige Anwendungsbeispiele gezeigt, in denen auch komplette Aspekte deklariert werden.

4.6 Aspekte in der Entwicklung

AspectJ wird oft zuerst während der Entwicklung von Programmen eingesetzt, um Vertrauen zur Sprache zu bekommen.

In der Entwicklung ist es oftmals notwendig, zur Fehlersuche zusätzliche Informationen auszugeben (Tracing, Logging), oder zur Programmoptimierung Zeitmessungen vorzunehmen (Profiling). Eine weitere Anforderung, die manche Sprachen nicht direkt unterstützen, ist das Überprüfen von Vor- und Nachbedingungen.

Gerade bei Debug–Ausgaben steckt der größte Aufwand im herausfinden der richtigen Stelle im Code. Wenn der Fehler gefunden wurde, ist dieser Code unnötig und wird oft wieder gelöscht oder auskommentiert. Dadurch ist diese Arbeit entweder ganz verloren, oder es macht den Code unübersichtlich. Werden diese Ausgaben in einem eigenen Aspekt gekapselt, sind sie schnell zu entfernen, aber auch sofort wieder integrierbar, wenn sie gebraucht werden.

Gerade bei diesen Anforderungen ist der Aspekt normalerweise über den gesamten Code verstreut; die Kapselung im Aspekt macht alles übersichtlicher.

Wenn das Produkt dann ausgeliefert werden soll, werden die Aspekte entfernt, und es bleibt kein Entwicklungs-Code versehentlich im Endprodukt.

Hier ein Beispiel, das verdeutlichen soll, wie fein man durch einen Aspekt Informationen erhalten kann:

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before():
    call(void Line.rotate(double)) {
      rotateCount++;
  }
```

```
  before():
    call(void Point.set*(int)) &&
    cflow(call(void Line.rotate
                          (double))) {
    setCount++;
  }
}
```

Dieser Aspekt zählt alle Aufrufe der Methode `rotate` aus der Klasse `Line`. Außerdem zählt er alle Aufrufe aller `set`–Methoden aus der Klasse `Point`. Allerdings nur die, die während der Ausführung der `rotate`–Methode aufgerufen werden.

Eine solche Einschränkung ist mit normalem Java nur sehr schwer möglich.

4.7 Aspekte im Produkt

AspectJ ist nicht nur in der Entwicklung zu gebrauchen.

Ein oft auftretendes Problem ist die Implementierung von Dirty–Flags:

Ein Editor soll z.B. einige Objekte in einem Fenster darstellen. Dazu löscht er das Fenster, durchläuft die Liste von Objekten und zeichnet jedes Objekt der Reihe nach. Jetzt sollen alle Objekte in einem Schritt 10 Pixel nach rechts verschoben werden. Dazu wird für jedes Objekt eine Methode aufgerufen, die die Position im Objekt ändert und dann das Fenster neu zeichnet.

Um diese Implementierung zu optimieren, kann ein Aspekt eingesetzt werden. Dieser merkt sich in einem Flag, ob irgendeines der dargestellten Objekte verändert wurde:

```
pointcut move():
  call(void *.setX(int)) ||
  call(void *.setY(int));

after() returning: move() {
  dirty = true;
}
```

Jetzt ist es einfach möglich, festzustellen, ob irgendein Objekt geändert wurde, und nur in diesem Fall genau einmal das Fenster zu löschen und die Objekte neu zu zeichnen.

Dies ist natürlich auch ohne AOP möglich, allerdings hat die Implementierung als Aspekt Vorteile:

- Der gesamte Code ist übersichtlich an einer Stelle
- Evtl. ist eine Erweiterung notwendig, einen sog. Aspektevolution; z.B. soll mit jeder Objektänderung auch das Objekt selbst in einer Liste gemerkt werden. Durch die Angabe mit einem einzigen Pointcut ist diese Änderung einfach und schnell möglich.
- Wenn diese Optimierung doch keinen Geschwindigkeitsvorteil bringt, kann man sie einfach wieder aus dem Programm herausnehmen
- Die Implementierung ist stabiler. Soll eine Methode in einer Unterklasse überschrieben werden, muß die Flag–Behandlung auch dorthin übernommen werden. Durch die Beschreibung im Aspekt kann das nicht vergessen werden: der Code wird automatisch mit–aufgerufen.

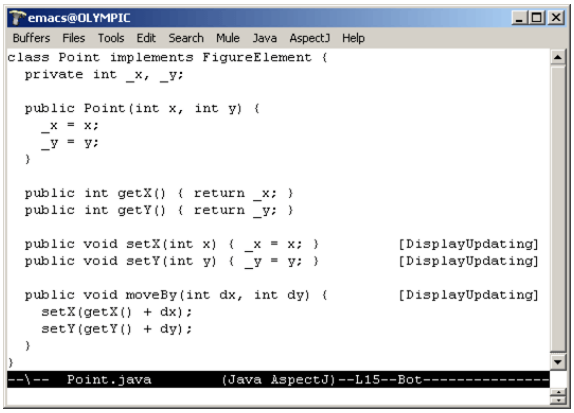

Abbildung 4: AspectJ im EMACS

Ein weiteres Anwendungsgebiet für Aspekte ist die Kapselung der Synchronisation.

4.8 Werkzeuge

Da AspectJ eine Erweiterung von Java ist, sind dafür speziell angepasste Werkzeuge notwendig.

Es ist ein Ersatz für `javac` verfügbar: `ajc`. Dieser ersetzt `javac` vollständig und erzeugt Java–Bytecode, der in jeder Standard–JVM läuft.

Außerdem gibt es Erweiterungen für Emacs (siehe Abb. 4), sowie die IDEs JBuilder und Forte.

Durch die Erweiterungen wird angezeigt, ob zu einer Methode Advices definiert sind und in welchem Aspekt diese stehen.

5 HyperJ

HyperJ hat einen etwas radikaleren Ansatz. Auch dabei geht es um die Kapselung sich überkreuzender Anforderungen; es wird dort *Multidimensional separation of concerns* (MDSOC) genannt. Die verschiedenen Anforderungen werden in *Hyperslices* gekapselt, die jeweils eine eigene Klassenhierarchie haben dürfen (aber nicht müssen).

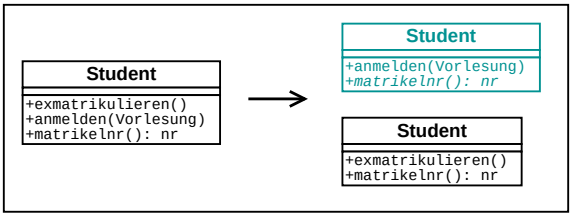

Abbildung 5: Remodularisierung mit HyperJ

Im Gegensatz zu AspectJ gibt es kein grundlegendes Klassen–Modell, in das verschiedene Aspekte eingewoben werden, sondern alle *Hyperslices* können sich gegenseitig überlagern. Auch HyperJ baut auf Java auf, wobei die Sprache selbst nicht geändert wurde, so daß alle Standard–Werkzeuge weiter verwendet werden können. Das weben wird durch zusätzliche Werkzeuge erreicht.

5.1 Remodularisierung

HyperJ unterstützt mit seinen Werkzeugen die Remodularisierung von Code, so daß eine Migration von Java zu HyperJ einfach möglich ist.

Dazu wird eine Datei erstellt, die definiert, welcher Code in welche Hyperslices verschoben werden soll.

Als Beispiel soll die Klasse `Student` in Abb. 5 remodularisiert werden.

```
package Personen:
  Feature.Verwaltung

operation anmelden:
  Feature.Vorlesungen
```

Mit dieser Datei werden die Operationen standardmäßig in den Hyperslice Verwaltung verschoben. Allerdings soll die Methode `anmelden` in den Hyperslice Vorlesungen. In unserem Beispiel benutzt `anmelden` die Methode `matrikelnr`. Daher wird diese als abstrakte Methode zusätzlich in den Hyperslice Vorlesungen übernommen. Die Implementierung verbleibt in Verwaltung.

5.2 Kombination

Um die Hyperslices Verwaltung und Vorlesungen wieder zu kombinieren, ist eine weitere Datei notwendig:

```
hypermodule VerwaltungUndVorlesungen
  hyperslices: Verwaltung,
               Vorlesungen;
  relationships: mergeByName;
end hypermodule;
```

Dadurch werden die Hyperslices wieder kombiniert, so daß die ursprüngliche Student–Klasse entsteht.

Diese Kombination wurde mit den Vergleich der Methodennamen durchgeführt. Es sind jedoch verschiedenste Kombinationsregeln möglich, so daß Methoden aus unterschiedlichen Hyperslices für eine Klasse zur Kombination benutzt werden können.

6 Subjektorientierte Programmierung (SOP)

Bei der subjektorientierten Programmierung[3] dreht sich alles um Objekte.

Ein Objekt hat innere Eigenschaften, wie z.B. seine Größe und seine Farbe, es kann faulen, wenn es ein Apfel ist. Außerdem gibt es äußere Eigenschaften, wie der Preis, oder daß man es auspacken kann.

Will man dies mit OOP abbilden, werden entweder alle Eigenschaften in einem Objekt vermischt, oder man benutzt Vererbung und kapselt alle Eigenschaften in einzelnen Klassen. Dies führt wieder zu den anfangs dargestellten Problemen.

SOP hat folgende Ansätze, um diesen zu begegnen:

- Anwendungen werden getrennt entwickelt und dann kombiniert
- es gibt keine festen Abhängigkeiten zwischen den Anwendungen
- es soll aber Kooperation zwischen ihnen möglich

Damit soll erreicht werden, daß sich neue Anwendungen leicht integrieren lassen.

Um diese Ziele zu erreichen, definiert die subjektorientierte Programmierung Subjekte.

6.1 Subjekte

Ein Subjekt besteht aus dem Status und dem Verhalten eines Objekts aus Sicht einer Anwendung. Es gibt dabei keine inneren Eigenschaften, diese können aber bei Bedarf als eigenes Subjekt modelliert werden. Kein Subjekt kennt das andere direkt; es ist nur eine Objekt–Identifikation (OID) bekannt. Diese Subjekte werden bei einem Aufruf in einer bestimmten Reihenfolge aktiviert. Das Verhalten eines Subjekts kann sich im zwischen den Aktivierungen ändern, im Gegensatz zur OOP, wo das Verhalten von der jeweiligen Klasse bestimmt ist.

Diese Idee lässt sich am Beispiel eines Baumes verdeutlichen. Für einen Vogel sind nur einzelne Eigenschaften eines Baumes interessant, z.B. ob er darin ein Nest bauen kann, wie hoch er ist, oder ob es in der Nähe genug Futter gibt. Einen Holzfäller interessieren andere Eigenschaften, z.B. auch die Höhe,

aber auch, ob der Baum verfault ist. Außerdem haben Vogel und Holzfäller unterschiedliche Verhaltensweisen, wenn der Baum gefällt wird. Der Holzfäller nimmt seine Kettensäge, der Vogel dagegen fliegt weg. Ein Subjekt ist die Kapselung dieser subjektiven Ansichten auf das Objekt Baum.

Sind die Subjekte isoliert, gibt es keinen Unterschied zur OOP. Daher ist eine Interaktion zwischen den Subjekten notwendig, die jedoch auch nicht zu eng sein soll. Subjekte interagieren mittels Komposition. Die Kompositionsregeln definieren, in welcher Reihenfolge die Subjekte aktiviert werden. Die Interfaces der verschiedenen Subjekte müssen dazu teilweise übereinstimmen.

Des weiteren ist es möglich, bei der Objekt–Instantiierung anzugeben, welche Instanzvariablen von mehreren Subjekten gemeinsam genutzt werden können.

7 Bewertung

Um den Nutzen und die Probleme dieser neuen Ansätze abschätzten zu können, sind Vergleiche zur OOP hilfreich.

In [4] ist eine Studie zum Vergleich verschiedener AOP-Techniken beschrieben. Es soll die Simulation eines Gebäude–Temperatur–Kontrollsystems programmiert werden. Implementierungssprache ist Java.

Vier verschiedene Ansätze werden verglichen:

- ein „traditioneller“ OO Ansatz
- AspectJ
- Reflective AO framework
- Event based architectural framework

7.1 Komplexität

In Abb. 6 werden verschiedene Codemetriken für die vier Ansätze dargestellt.

Auffällig ist, daß der OO-Ansatz nur halb so viele Klassen wie die anderen Ansätze hat, dafür sind die Klassen und die Methoden größer. Somit wird das Versprechen der AOP, den Code besser zu modularisieren, hier eingehalten.

Allerdings ist der Code der AOP-Lösungen 50% größer.

7.2 Performance

Abb. 7 zeigt die Laufzeit der OO-Implementierung und relativ dazu die Zeiten der anderen Ansätze.

Es fällt vor allem auf, daß das auf Reflektion basierende Framework die dreifache Laufzeit hat und

Metrik	OO	AspectJ	Refl.	a. FW
Klassen	7	14	15	21
Methoden	143	213	232	263
NCSS	969	1354	1486	1605
Methoden/Kl.	20,43	15,21	15,47	12,52
NCSS/Kl.	138,43	96,71	99,07	76,43
NCSS/Met.	6,78	6,36	6,41	6,10
Ø CCN / Meth.	2,64	2,51	2,51	2,32

NCSS = Non Commenting Source Statements
CCN = Zyklomatische Komplexität nach McCabe

Abbildung 6: Codemetriken

Iterationen	OO [msec]	AspectJ
100	54762	100.93%
1000	515420	100.85%
2000	1024309	100.22%
5000	2539091	100.34%
Iterationen	Reflective	Arch. FW
100	330.91%	101.15%
1000	348.17%	100.47%
2000	348.41%	100.06%
5000	354.17%	99.94%

Abbildung 7: Laufzeiten

mit der Anzahl der Iterationen sogar noch zunimmt. Die anderen Implementierungen sind dagegen konkurrenzfähig, der Unterschied nimmt mit der Laufzeit sogar noch ab.

Literatur

[1] ASPECTJ.ORG: *The AspectJ Primer*.

[2] BÖLLERT, KAI: *On Weaving Aspects*. 1999.

[3] HARRISON, W. und H. OSSER: *Subject-Oriented Programming (A critique of Pure Objects)*. ACM SIGPLAN Notices, 28(10):411-428, October 1993.

[4] J. ANDRES DIAZ PACE, MARCELO R. CAMPO: *Analyzing the role of aspects in software design*. Communications of the ACM, 44:67-73, October 2001.

[5] KICZALES, G., ET AL.: *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[6] KICZALES, G., ET AL.: *An overview of AspectJ*. Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), 2001.